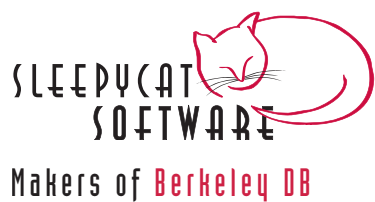


Berkeley DB Java Edition Transaction Processing



Legal Notice

This documentation is distributed under the terms of the Sleepycat public license. You may review the terms of this license at: <http://www.sleepycat.com/download/oslicense.html>

Sleepycat Software, Berkeley DB, Berkeley DB XML and the Sleepycat logo are trademarks or service marks of Sleepycat Software, Inc. All rights to these marks are reserved. No third-party use is permitted without the express prior written consent of Sleepycat Software, Inc.

Java™ and all Java-based marks are a trademark or registered trademark of Sun Microsystems, Inc, in the United States and other countries.

To obtain a copy of this document's original source code, please write to <support@sleepycat.com>.

Published 5/24/2006

Table of Contents

Preface	iv
Conventions Used in this Book	iv
For More Information	v
1. Introduction	1
Transaction Benefits	1
A Note on System Failure	2
Application Requirements	2
Multi-threaded Applications	3
Recoverability	4
Performance Tuning	4
2. Enabling Transactions	5
Opening a Transactional Environment and Database	5
3. Transaction Basics	7
Committing a Transaction	8
Non-Durable Transactions	9
Aborting a Transaction	10
Auto Commit	11
Transactional Cursors	12
Secondary Indices with Transaction Applications	13
Configuring the Transaction Subsystem	15
4. Concurrency	17
Which JE Handles are Free-Threaded	18
Locks, Blocks, and Deadlocks	18
Locks	18
Lock Resources	19
Types of Locks	19
Lock Lifetime	20
Blocks	20
Blocking and Application Performance	21
Avoiding Blocks	22
Deadlocks	22
Deadlock Avoidance	23
JE Lock Management	24
Managing JE Lock Timeouts	24
Managing Deadlocks	24
Isolation	26
Supported Degrees of Isolation	26
Reading Uncommitted Data	27
Committed Reads	29
Configuring Serializable Isolation	31
Transactional Cursors and Concurrent Applications	33
Using Cursors with Uncommitted Data	34
Read/Modify/Write	35
5. Backing up and Restoring Berkeley DB, Java Edition Applications	37
Normal Recovery	37
Checkpoints	37

Performing Backups	38
Performing a Hot Backup	38
Performing an Offline Backup	38
Using the DbBackup Helper Class	39
Performing Catastrophic Recovery	41
Hot Failover	41
6. Summary and Examples	43
Anatomy of a Transactional Application	43
Transaction Example	44
TxnGuide.java	45
PayloadData.java	48
DBWriter.java	49

Preface

This document describes how to use transactions with your Berkeley DB, Java Edition applications. It is intended to describe how to transaction protect your application's data. The APIs used to perform this task are described here, as are the environment infrastructure and administrative tasks required by a transactional application. This book also describes multi-threaded JE applications and the requirements they have for deadlock detection.

This book is aimed at the software engineer responsible for writing a transactional JE application.

This book assumes that you have already read and understood the concepts contained in *Getting Started with Berkeley DB, Java Edition*.

Conventions Used in this Book

The following typographical conventions are used within in this manual:

Class names are represented in monospaced font, as are method names. For example: "The `Environment.openDatabase()` method returns a `Database` class object."

Variable or non-literal text is presented in *italics*. For example: "Go to your *JE_HOME* directory."

Program examples are displayed in a monospaced font on a shaded background. For example:

```
import com.sleepycat.je.Environment;

...

// Open the environment. Allow it to be created if it does not already exist.
Environment myDbEnv;
```

In some situations, programming examples are updated from one chapter to the next. When this occurs, the new code is presented in monospaced bold font. For example:

```
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import java.io.File;

...

// Open the environment. Allow it to be created if it does not already exist.
Environment myDbEnv;
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setAllowCreate(true);
myDbEnv = new Environment(new File("/export/dbEnv"), envConfig);
```



Finally, notes of special interest are represented using a note block such as this.

For More Information

Beyond this manual, you may also find the following sources of information useful when building a transactional JE application:

- [Berkeley DB Java Edition Getting Started Guide](http://www.sleepycat.com/jedocs/GettingStartedGuide/BerkeleyDB-JE-GSG.pdf)
[<http://www.sleepycat.com/jedocs/GettingStartedGuide/BerkeleyDB-JE-GSG.pdf>]
- [Berkeley DB Java Edition Javadoc](http://www.sleepycat.com/jedocs/java/index.html) [<http://www.sleepycat.com/jedocs/java/index.html>]
- [Berkeley DB, Java Edition Getting Started with the Direct Persistence Layer](http://www.sleepycat.com/jedocs/PersistenceAPI/BerkeleyDB-JE-Persistence-GSG.pdf)
[<http://www.sleepycat.com/jedocs/PersistenceAPI/BerkeleyDB-JE-Persistence-GSG.pdf>]

Chapter 1. Introduction

This book provides a thorough introduction and discussion on transactions as used with Berkeley DB, Java Edition (JE). It begins by offering a general overview to transactions, the guarantees they provide, and the general application infrastructure required to obtain full transactional protection for your data.

This book also provides detailed examples on how to write a transactional application. Both single threaded and multi-threaded are discussed. A detailed description of various backup and recovery strategies is included in this manual, as is a discussion on performance considerations for your transactional application.

You should understand the concepts from *Getting Started with Berkeley DB, Java Edition* before reading this book.



The examples presented in this book use the Berkeley DB, Java Edition API, not the Direct Persistence Layer API. However, all of the concepts presented here in terms of transactional benefits, transaction usage, and deadlock handling also apply to the Direct Persistence Layer.

Transaction Benefits

Transactions offer your application's data protection from application or system failures. That is, JE transactions offer your application full ACID support:

- Atomicity

Multiple database operations are treated as a single unit of work. Once committed, all write operations performed under the protection of the transaction are saved to your databases. Further, in the event that you abort a transaction, all write operations performed during the transaction are discarded. In this event, your database is left in the state it was in before the transaction began, regardless of the number or type of write operations you may have performed during the course of the transaction.

Note that JE transactions can span one or more database handles.

- Consistency

Your databases will never see a partially completed transaction. This is true even if your application fails while there are in-progress transactions. If the application or system fails, then either all of the database changes appear when the application next runs, or none of them appear.

In other words, whatever consistency requirements your application has will never be violated by JE. If, for example, your application requires every record to include an employee ID, and your code faithfully adds that ID to its database records, then JE will never violate that consistency requirement. The ID will remain in the database records until such a time as your application chooses to delete it.

- Isolation

While a transaction is in progress, your databases will appear to the transaction as if there are no other operations occurring outside of the transaction. That is, operations wrapped inside a transaction will always have a clean and consistent view of your databases. They never have to see updates currently in progress under the protection of another transaction. Note, however, that isolation guarantees can be increased and relaxed from the default setting. See [Isolation \(page 26\)](#) for more information.

- **Durability**

Once committed to your databases, your modifications will persist even in the event of an application or system failure. Note that like isolation, your durability guarantee can be relaxed. See [Non-Durable Transactions \(page 9\)](#) for more information.

A Note on System Failure

From time to time this manual mentions that transactions protect your data against 'system or application failure.' This is true up to a certain extent. However, not all failures are created equal and no data protection mechanism can protect you against every conceivable way a computing system can find to die.

Generally, when this book talks about protection against failures, it means that transactions offer protection against the likeliest culprits for system and application crashes. So long as your data modifications have been committed to disk, those modifications should persist even if your application or OS subsequently fails. And, even if the application or OS fails in the middle of a transaction commit (or abort), the data on disk should be either in a consistent state, or there should be enough data available to bring your databases into a consistent state (via a recovery procedure, for example). You may, however, lose whatever data you were committing at the time of the failure, but your databases will be otherwise unaffected.

Of course, if your *disk* fails, then the transactional benefits described in this book are only as good as the backups you have taken.

Finally, by following the programming examples shown in this book, you can write your code so as to protect your data in the event that your code crashes. However, no programming API can protect you against logic failures in your own code; transactions cannot protect you from simply writing the wrong thing to your databases.

Application Requirements

In order to use transactions, your application has certain requirements beyond what is required of non-transactional protected applications. They are:

- Transaction subsystem.

In order to use transactions, you must explicitly enable the transactional subsystem for your application, and this must be done at the time that your environment is first created.

- Transaction handles.

In order to obtain the atomicity guarantee offered by the transactional subsystem (that is, combine multiple operations in a single unit of work), your application must use transaction handles. These handles are obtained from your Environment objects. They should normally be short-lived, and their usage is reasonably simple. To complete a transaction and save the work it performed, you call its `commit()` method. To complete a transaction and discard its work, you call its `abort()` method.

In addition, it is possible to use auto commit if you want to transactionally protect a single write operation. Auto commit allows a transaction to be used without obtaining an explicit transaction handle. See [Auto Commit \(page 11\)](#) for information on how to use auto commit.

- Database open requirements.

Your application must transactionally protect the database opens, and any secondary index associations, if subsequent operations on the databases are to be transactionally protected. The database open and secondary index association are commonly transactionally protected using auto commit.

- Deadlock detection.

Typically transactional applications use multiple threads of control when accessing the database. Any time multiple threads are used on a single resource, the potential for lock contention arises. In turn, lock contention can lead to deadlocks. See [Locks, Blocks, and Deadlocks \(page 18\)](#) for more information.

Therefore, transactional applications must frequently include code for detecting and responding to deadlocks. Note that this requirement is not *specific* to transactions - you can certainly write concurrent non-transactional JE applications. Further, not every transactional application uses concurrency and so not every transactional application must manage deadlocks. Still, deadlock management is so frequently a characteristic of transactional applications that we discuss it in this book. See [Concurrency \(page 17\)](#) for more information.

Multi-threaded Applications

JE is designed to support multi-threaded applications, but their usage means you must pay careful attention to issues of concurrency. Transactions help your application's concurrency by providing various levels of isolation for your threads of control. In addition, JE provides mechanisms that allow you to detect and respond to deadlocks (but strictly speaking, this is not limited to just transactional applications).

Isolation means that database modifications made by one transaction will not normally be seen by readers from another transaction until the first commits its changes. Different threads use different transaction handles, so this mechanism is normally used to provide isolation between database operations performed by different threads.

Note that JE supports different isolation levels. For example, you can configure your application to see uncommitted reads, which means that one transaction can see data that has been modified but not yet committed by another transaction. Doing this might

mean your transaction reads data "dirty" by another transaction, but which subsequently might change before that other transaction commits its changes. On the other hand, lowering your isolation requirements means that your application can experience improved throughput due to reduced lock contention.

For more information on concurrency, on managing isolation levels, and on deadlock detection, see [Concurrency \(page 17\)](#).

Recoverability

An important part of JE's transactional guarantees is durability. *Durability* means that once a transaction has been committed, the database modifications performed under its protection will not be lost due to system failure.

JE supports a normal recovery that runs against a subset of your log files. This is a routine procedure used whenever your environment is first opened upon application startup, and it is intended to ensure that your database is in a consistent state. JE also supports archival backup and recovery in the case of catastrophic failure, such as the loss of a physical disk drive.

This book describes several different backup procedures you can use to protect your on-disk data. These procedures range from simple offline backup strategies to hot failovers. Hot failovers provide not only a backup mechanism, but also a way to recover from a fatal hardware failure.

This book also describes the recovery procedures you should use for each of the backup strategies that you might employ.

For a detailed description of backup and restore procedures, see the [Berkeley DB Java Edition Getting Started Guide](#).

[<http://www.sleepycat.com/jedocs/GettingStartedGuide/BerkeleyDB-JE-GSG.pdf>]

Performance Tuning

From a performance perspective, the use of transactions is not free. Depending on how you configure them, transaction commits usually require your application to perform disk I/O that a non-transactional application does not perform. Also, for multi-threaded applications, the use of transactions can result in increased lock contention due to extra locking requirements driven by transactional isolation guarantees.

There is therefore a performance tuning component to transactional applications that is not applicable for non-transactional applications (although some tuning considerations do exist whether or not your application uses transactions). Where appropriate, these tuning considerations are introduced in the following chapters.

Chapter 2. Enabling Transactions

In order to use transactions with your application, you must turn them on. To do this you must:

- Turn on transactions for your environment. You do this by using the `EnvironmentConfig.setTransactional()` method, or by using the `je.env.isTransactional je.properties` parameter.
- Transaction-enable your databases. You do this by using the `DatabaseConfig.setTransactional()` method, and then opening the database from within a transaction. Note that the common practice is for auto commit to be used to transaction-protect the database open. To use auto-commit, you must still enable transactions as described here, but you do not have to explicitly use a transaction when you open your database. An example of this is given in the next section.

Opening a Transactional Environment and Database

To enable transactions for your environment, you must initialize the transactional subsystem:

```
package je.txn;

import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;

import java.io.File;

...

Environment myEnv = null;
try {
    EnvironmentConfig myEnvConfig = new EnvironmentConfig();
    myEnvConfig.setTransactional(true);

    myEnv = new Environment(new File("/my/env/home"),
                           myEnvConfig);

} catch (DatabaseException de) {
    // Exception handling goes here
}
```

You then create and open your database(s) as you would for a non-transactional system. The only difference is that you must set `DatabaseConfig.setTransactional()` to `true`. Note that your database open must be transactional-protected. However, if you do not give the `openDatabase()` method a transaction handle, then the open is automatically protected using auto commit. Typically auto commit is used for this purpose. For example:

```
package je.txn;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;

import java.io.File;

...

Database myDatabase = null;
Environment myEnv = null;
try {
    EnvironmentConfig myEnvConfig = new EnvironmentConfig();
    myEnvConfig.setTransactional(true);
    myEnv = new Environment(new File("/my/env/home"),
                           myEnvConfig);

    // Open the database. Create it if it does not already exist.
    DatabaseConfig dbConfig = new DatabaseConfig();
    dbConfig.setTransactional(true);
    myDatabase = myEnv.openDatabase(null,
                                    "sampleDatabase",
                                    dbConfig);

} catch (DatabaseException de) {
    // Exception handling goes here
}
```



Never close a database that has active transactions. Make sure all transactions are resolved (either committed or aborted) before closing the database.

Chapter 3. Transaction Basics

Once you have enabled transactions for your environment and your databases, you can use them to protect your database operations. You do this by acquiring a transaction handle and then using that handle for any database operation that you want to participate in that transaction.

You obtain a transaction handle using the `Environment.beginTransaction()` method.

Once you have completed all of the operations that you want to include in the transaction, you must commit the transaction using the `Transaction.commit()` method.

If, for any reason, you want to abandon the transaction, you abort it using `Transaction.abort()`.

Any transaction handle that has been committed or aborted can no longer be used by your application.

Finally, you must make sure that all transaction handles are either committed or aborted before closing your databases and environment.



If you only want to transaction protect a single database write operation, you can use auto commit to perform the transaction administration. When you use auto commit, you do not need an explicit transaction handle. See [Auto Commit \(page 11\)](#) for more information.

For example, the following example opens a transactional-enabled environment and database, obtains a transaction handle, and then performs a write operation under its protection. In the event of any failure in the write operation, the transaction is aborted and the database is left in a state as if no operations had ever been attempted in the first place.

```
package je.txn;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import com.sleepycat.je.Transaction;

import java.io.File;

...

Database myDatabase = null;
Environment myEnv = null;
try {
    EnvironmentConfig myEnvConfig = new EnvironmentConfig();
    myEnvConfig.setTransactional(true);
```

```
myEnv = new Environment(new File("/my/env/home"),
                        myEnvConfig);

// Open the database. Create it if it does not already exist.
DatabaseConfig dbConfig = new DatabaseConfig();
dbConfig.setTransactional(true);
myDatabase = myEnv.openDatabase(null,
                                "sampleDatabase",
                                dbConfig);

String keyString = "thekey";
String dataString = "thedata";
DatabaseEntry key =
    new DatabaseEntry(keyString.getBytes("UTF-8"));
DatabaseEntry data =
    new DatabaseEntry(dataString.getBytes("UTF-8"));

Transaction txn = myEnv.beginTransaction(null, null);

try {
    myDatabase.put(txn, key, data);
    txn.commit();
} catch (Exception e) {
    if (txn != null) {
        txn.abort();
        txn = null;
    }
}

} catch (DatabaseException de) {
    // Exception handling goes here
}
```

Committing a Transaction

In order to fully understand what is happening when you commit a transaction, you must first understand a little about what JE is doing with its log files. Logging causes all database write operations to be identified in log files (remember that in JE, your log files *are* your database files; there is no difference between the two). Enough information is written to restore your entire BTree in the event of a system or application failure, so by performing logging, JE ensures the integrity of your data.

Remember that all write activity made to your database is identified in JE's logs as the writes are performed by your application. However, JE maintains logs in memory. Eventually this information is written to disk, but especially in the case of a transactional application this data may be held in memory until the transaction is committed, or JE runs out of buffer space for the logging information.

When you commit a transaction, the following occurs:

- A commit record is written to the log. This indicates that the modifications made by the transaction are now permanent. By default, this write is performed synchronously to disk so the commit record arrives in the log files before any other actions are taken.
- Any log information held in memory is (by default) synchronously written to disk. Note that this requirement can be relaxed, depending on the type of commit you perform. See [Non-Durable Transactions \(page 9\)](#) for more information.

Note that a transaction commit only writes the BTree's leaf nodes to JE's log files. All other internal BTree structures are left unwritten.

- All locks held by the transaction are released. This means that read operations performed by other transactions or threads of control can now see the modifications without resorting to uncommitted reads (see [Reading Uncommitted Data \(page 27\)](#) for more information).

To commit a transaction, you simply call `Transaction.commit()`.

Remember that transaction commit causes only the BTree leaf nodes to be written to JE's log files. Any other modifications made to the the BTree as a result of the transaction's activities are not written to the log file. This means that over time JE's normal recovery time can greatly increase (remember that JE always runs normal recovery when it opens an environment).

For this reason, JE by default runs the checkpoint thread. This background thread runs a checkpoint on a periodic interval so as to ensure that the amount of data that needs to be recovered upon environment open is minimized. In addition, you can also run a checkpoint manually. For more information, see [Checkpoints \(page 37\)](#).

Note that once you have committed a transaction, the transaction handle that you used for the transaction is no longer valid. To perform database activities under the control of a new transaction, you must obtain a fresh transaction handle.

Non-Durable Transactions

As previously noted, by default transaction commits are durable because they cause the modifications performed under the transaction to be synchronously recorded in your on-disk log files. However, it is possible to use non-durable transactions.

You may want non-durable transactions for performance reasons. For example, you might be using transactions simply for the isolation guarantee. In this case, you might want to relax the synchronized write to disk that JE normally performs as part of a transaction commit. Doing so means that your data will still make it to disk; however, your application will not necessarily have to wait for the disk I/O to complete before it can perform another database operation. This can greatly improve throughput for some workloads.

There are several ways to relax the synchronized write requirement for your transactions:

- Specify `true` to the `EnvironmentMutableConfig.setTxnNoSync()` method. This causes JE to not synchronously force any data to disk upon transaction commit. That is, the

modifications are held entirely inside the JVM and the modifications are not forced to the file system for long-term storage. Note, however, that the data will eventually make it to the filesystem (assuming no application or OS crashes) as a part of JE's management of its logging buffers and/or cache.

This form of a commit provides a weak durability guarantee because data loss can occur due to an application, JVM, or OS crash.

This behavior is specified on a per-environment handle basis. In order for your application to exhibit consistent behavior, you need to specify this for all of the environment handles used in your application.

You can achieve this behavior on a transaction by transaction basis by using `Transaction.commitNoSync()` to commit your transaction, or by specifying `true` to the `TransactionConfig.setNoSync()` method when starting the transaction.

- Specify `true` to the `EnvironmentConfig.setTxnWriteNoSync()` method. This causes data to be synchronously written to the OS's file system buffers upon transaction commit. The data will eventually be written to disk, but this occurs when the operating system chooses to schedule the activity; the transaction commit can complete successfully before this disk I/O is performed by the OS.

This form of commit protects you against application and JVM crashes, but not against OS crashes. This method offers less room for the possibility of data loss than does `EnvironmentConfig.setTxnNoSync()`.

This behavior is specified on a per-environment handle basis. In order for your application to exhibit consistent behavior, you need to specify this for all of the environment handles used in your application.

You can achieve this behavior on a transaction by transaction basis by using `Transaction.commitWriteNoSync()` to commit your transaction, or by specifying `true` to `TransactionConfig.setWriteNoSync()` method when starting the transaction.

Aborting a Transaction

When you abort a transaction, all database modifications performed under the protection of the transaction are discarded, and all locks currently held by the transaction are released. In this event, your data is simply left in the state that it was in before the transaction began performing data modifications.

Note that aborting a transaction may result in disk I/O. It is possible that during the course of your transaction, logging data and/or database records were written to backing files on disk. For this reason, JE notes that the abort occurred in its log files so that at a minimum the database can be brought into a consistent state at recovery time.

Also, once you have aborted a transaction, the transaction handle that you used for the transaction is no longer valid. To perform database activities under the control of a new transaction, you must obtain a fresh transactional handle.

To abort a transaction, call `Transaction.abort()`.

Auto Commit

While transactions are frequently used to provide atomicity to multiple database operations, it is sometimes necessary to perform a single database operation under the control of a transaction. Rather than force you to obtain a transaction, perform the single write operation, and then either commit or abort the transaction, you can automatically group this sequence of events using *auto commit*.

To use auto commit:

1. Open your environment and your databases so that they support transactions. See [Enabling Transactions \(page 5\)](#) for details.
2. Do not provide a transactional handle to the method that is performing the database write operation.

Note that auto commit is not available for cursors. You must always open your cursor using a transaction if you want the cursor's operations to be transactional protected. See [Transactional Cursors \(page 12\)](#) for details on using transactional cursors.

For example, the following uses auto commit to perform the database write operation:

```
package je.txn;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;

import java.io.File;

...

Database myDatabase = null;
Environment myEnv = null;
try {
    EnvironmentConfig myEnvConfig = new EnvironmentConfig();
    myEnvConfig.setTransactional(true);
    myEnv = new Environment(new File("/my/env/home"),
                           myEnvConfig);

    // Open the database. Create it if it does not already exist.
    DatabaseConfig dbConfig = new DatabaseConfig();
    dbConfig.setTransactional(true);
    myDatabase = myEnv.openDatabase(null,
                                    "sampleDatabase",
```

```

dbConfig);

String keyString = "thekey";
String dataString = "thedata";
DatabaseEntry key =
    new DatabaseEntry(keyString.getBytes("UTF-8"));
DatabaseEntry data =
    new DatabaseEntry(dataString.getBytes("UTF-8"));

// Perform the write. Because the database was opened to
// support transactions, this write is performed using auto commit.
myDatabase.put(null, key, data);

} catch (DatabaseException de) {
    // Exception handling goes here
}

```

Transactional Cursors

You can transaction-protect your cursor operations by specifying a transaction handle at the time that you create your cursor. Beyond that, you do not ever provide a transaction handle directly to a cursor method.

Note that if you transaction-protect a cursor, then you must make sure that the cursor is closed before you either commit or abort the transaction. For example:

```

package je.txn;

import com.sleepycat.je.Cursor;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;
import com.sleepycat.je.Transaction;

import java.io.File;

...

Database myDatabase = null;
Environment myEnv = null;
try {

    // Database and environment opens omitted

```

```
String replacementData = "new data";

Transaction txn = myEnv.beginTransaction(null, null);
Cursor cursor = null;
try {
    // Use the transaction handle here
    cursor = db.openCursor(txn, null);
    DatabaseEntry key, data;

    DatabaseEntry key, data;
    while(cursor.getNext(key, data, LockMode.DEFAULT) ==
        OperationStatus.SUCCESS) {

        data.setData(replacementData.getBytes("UTF-8"));
        // No transaction handle is used on the cursor read or write
        // methods.
        cursor.putCurrent(data);
    }

    cursor.close();
    cursor = null;
    txn.commit();
} catch (Exception e) {
    if (cursor != null) {
        cursor.close();
    }
    if (txn != null) {
        txn.abort();
        txn = null;
    }
}

} catch (DatabaseException de) {
    // Exception handling goes here
}
```

Secondary Indices with Transaction Applications

You can use transactions with your secondary indices so long as you open the secondary index so that it is transactional.

All other aspects of using secondary indices with transactions are identical to using secondary indices without transactions. In addition, transaction-protecting secondary cursors is performed just as you protect normal cursors — you simply have to make sure the cursor is opened using a transaction handle, and that the cursor is closed before the handle is either committed or aborted. See [Transactional Cursors \(page 12\)](#) for details.

Note that when you use transactions to protect your database writes, your secondary indices are protected from corruption because updates to the primary and the secondaries are performed in a single atomic transaction.

For example:

```
package je.txn;

import com.sleepycat.bind.tuple.TupleBinding;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseType;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import com.sleepycat.je.SecondaryDatabase;
import com.sleepycat.je.SecondaryConfig;

import java.io.FileNotFoundException;

...

// Environment and primary database opens omitted.

SecondaryConfig mySecConfig = new SecondaryConfig();
mySecConfig.setAllowCreate(true);
mySecConfig.setTransactional(true);

SecondaryDatabase mySecDb = null;
try {
    // A fake tuple binding that is not actually implemented anywhere.
    // The tuple binding is dependent on the data in use.
    // See the Getting Started Guide for details
    TupleBinding myTupleBinding = new MyTupleBinding();

    // Open the secondary. FullNameKeyCreator is not actually implemented
    // anywhere. See the Getting Started Guide for details.
    FullNameKeyCreator keyCreator = new FullNameKeyCreator(myTupleBinding);

    // Set the key creator on the secondary config object.
    mySecConfig.setKeyCreator(keyCreator);

    // Perform the actual open. Because this database is configured to be
    // transactional, the open is automatically wrapped in a transaction.
    //     - myEnv is the environment handle.
    //     - myDb is the primary database handle.
    String secDbName = "mySecondaryDatabase";
    mySecDb = myEnv.openSecondary(null, secDbName, null, myDb, mySecConfig);
} catch (DatabaseException de) {
```

```
// Exception handling goes here ...  
}
```

Configuring the Transaction Subsystem

When you configure your transaction subsystem, you need to consider your transaction timeout value. This value represents the longest period of time a transaction can be active. Note, however, that transaction timeouts are checked only when JE examines its lock tables for blocked locks (see [Locks, Blocks, and Deadlocks \(page 18\)](#) for more information). Therefore, a transaction's timeout can have expired, but the application will not be notified until JE has a reason to examine its lock tables.

Be aware that some transactions may be inappropriately timed out before the transaction has a chance to complete. You should therefore use this mechanism only if you know your application might have unacceptably long transactions and you want to make sure your application will not stall during their execution. (This might happen if, for example, your transaction blocks or requests too much data.)

Note that by default transaction timeouts are set to 0 seconds, which means that they never time out.

To set the maximum timeout value for your transactions, use the `EnvironmentConfig.setTxnTimeout()` method. This method configures the entire environment; not just the handle used to set the configuration. Further, this value may be set at any time during the application's lifetime.

This value can also be set using the `je.txn.timeout` property in your JE properties file.

For example:

```
package je.txn;  
  
import com.sleepycat.je.Environment;  
import com.sleepycat.je.EnvironmentConfig;  
  
import java.io.File;  
  
...  
  
Environment myEnv = null;  
try {  
    EnvironmentConfig myEnvConfig = new EnvironmentConfig();  
    myEnvConfig.setTransactional(true);  
  
    // Configure a maximum transaction timeout of 1 second.  
    myEnvConfig.setTxnTimeout(1000000);  
  
    myEnv = new Environment(new File("/my/env/home"),  
                           myEnvConfig);  
}
```

```
// From here, you open your databases, proceed with your  
// database operations, and respond to deadlocks as  
// is normal (omitted for brevity).
```

```
...
```

Chapter 4. Concurrency

JE offers a great deal of support for multi-threaded applications even when transactions are not in use. Many of JE's handles are thread-safe and JE provides a flexible locking subsystem for managing databases in a concurrent application. Further, JE provides a robust mechanism for detecting and responding to deadlocks. All of these concepts are explored in this chapter.

Before continuing, it is useful to define a few terms that will appear throughout this chapter:

- *Thread of control*

Refers to a thread that is performing work in your application. Typically, in this book that thread will be performing JE operations.

- *Locking*

When a thread of control obtains access to a shared resource, it is said to be *locking* that resource. Note that JE supports both exclusive and non-exclusive locks. See [Locks \(page 18\)](#) for more information.

- *Free-threaded*

Data structures and objects are free-threaded if they can be shared across threads of control without any explicit locking on the part of the application. Some books, libraries, and programming languages may use the term *thread-safe* for data structures or objects that have this characteristic. The two terms mean the same thing.

For a description of free-threaded JE objects, see [Which JE Handles are Free-Threaded \(page 18\)](#).

- *Blocked*

When a thread cannot obtain a lock because some other thread already holds a lock on that object, the lock attempt is said to be *blocked*. See [Blocks \(page 20\)](#) for more information.

- *Deadlock*

Occurs when two or more threads of control attempt to access conflicting resource in such a way as none of the threads can any longer may further progress.

For example, if Thread A is blocked waiting for a resource held by Thread B, while at the same time Thread B is blocked waiting for a resource held by Thread A, then neither thread can make any forward progress. In this situation, Thread A and Thread B are said to be *deadlocked*.

For more information, see [Deadlocks \(page 22\)](#).

Which JE Handles are Free-Threaded

The following describes to what extent and under what conditions individual handles are free-threaded.

- `Environment`

This class is free-threaded.

- `Database`

This class is free-threaded.

- `SecondaryDatabase`

This class is free-threaded.

- `Cursor`


If the cursor is a transactional cursor, it can be used by multiple threads of control so long as the application serializes access to the handle. If the cursor is not a transactional cursor, it can not be shared across multiple threads of control at all.

- `SecondaryCursor`

Same conditions apply as for `Cursor` handles.

- `Transaction`

This class is free-threaded.

 All classes found in the bind APIs (`com.sleepycat.bind.*`) are free-threaded.

Locks, Blocks, and Deadlocks

It is important to understand how locking works in a concurrent application before continuing with a description of the concurrency mechanisms JE makes available to you. Blocking and deadlocking have important performance implications for your application. Consequently, this section provides a fundamental description of these concepts, and how they affect JE operations.

Locks

When one thread of control wants to obtain access to an object, it requests a *lock* for that object. This lock is what allows JE to provide your application with its transactional isolation guarantees by ensuring that:

- no other thread of control can read that object (in the case of an exclusive lock), and

- no other thread of control can modify that object (in the case of an exclusive or non-exclusive lock).

Lock Resources

When locking occurs, there are conceptually three resources in use:

1. The locker.

This is the thing that holds the lock. In a transactional application, the locker is a transaction handle. For non-transactional operations, the locker is the current thread.

2. The lock.

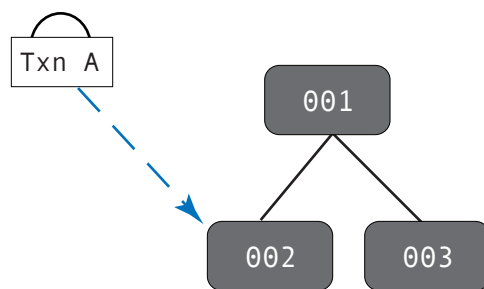
This is the actual data structure that locks the object. In JE, a locked object structure in the lock manager is representative of the object that is locked.

3. The locked object.

The thing that your application actually wants to lock. In a JE application, the locked object is usually a database record.

JE has not set a limit for the maximum number of these resources you can use. Instead, you are only limited by the amount of memory available to your application.

The following figure shows a transaction handle, `Txn A`, that is holding a lock on database record 002. In this graphic, `Txn A` is the locker, and the locked object is record 002. Only a single lock is in use in this operation.



Types of Locks

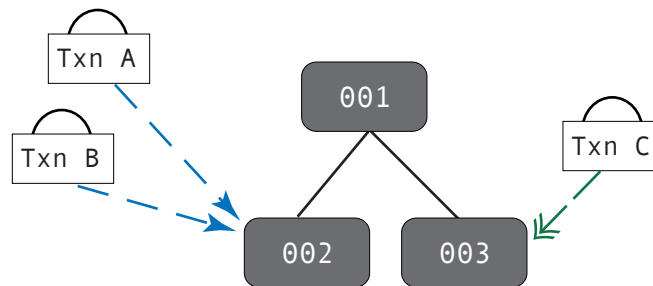
JE applications support both exclusive and non-exclusive locks. *Exclusive locks* are granted when a locker wants to write to an object. For this reason, exclusive locks are also sometimes called *write locks*.

An exclusive lock prevents any other locker from obtaining any sort of a lock on the object. This provides isolation by ensuring that no other locker can observe or modify an exclusively locked object until the locker is done writing to that object.

Non-exclusive locks are granted for read-only access. For this reason, non-exclusive locks are also sometimes called *read locks*. Since multiple lockers can simultaneously hold read locks on the same object, read locks are also sometimes called *shared locks*.

A non-exclusive lock prevents any other locker from modifying the locked object while the locker is still reading the object. This is how transactional cursors are able to achieve repeatable reads; by default, the cursor's transaction holds a read lock on any object that the cursor has examined until such a time as the transaction is committed or aborted.

In the following figure, Txn A and Txn B are both holding read locks on record 002, while Txn C is holding a write lock on record 003:



Lock Lifetime

A locker holds its locks until such a time as it does not need the lock any more. What this means is:

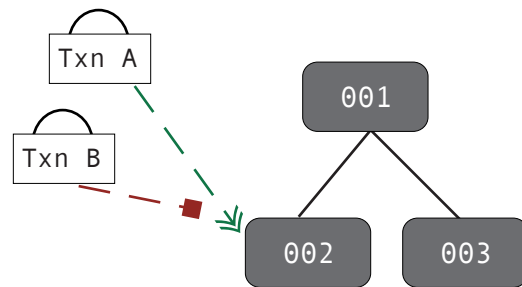
1. A transaction holds any locks that it obtains until the transaction is committed or aborted.
2. All non-transaction operations hold locks until such a time as the operation is completed. For cursor operations, the lock is held until the cursor is moved to a new position or closed.

Blocks

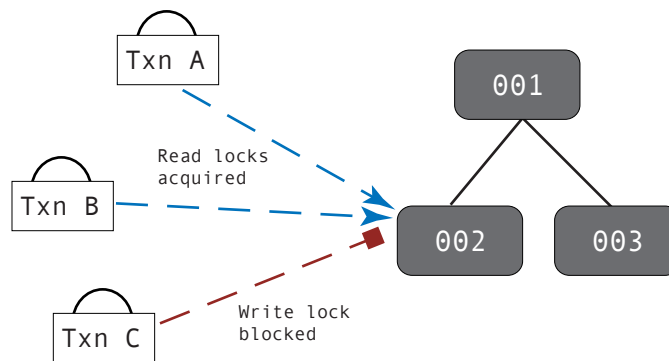
Simply put, a thread of control is blocked when it attempts to obtain a lock, but that attempt is denied because some other thread of control holds a conflicting lock. Once blocked, the thread of control is temporarily unable to make any forward progress until the requested lock is obtained or the operation requesting the lock is abandoned.

Be aware that when we talk about blocking, strictly speaking the thread is not what is attempting to obtain the lock. Rather, some object within the thread (such as a cursor) is attempting to obtain the lock. However, once a locker attempts to obtain a lock, the entire thread of control must pause until the lock request is in some way resolved.

For example, if Txn A holds a write lock (an exclusive lock) on record 002, then if Txn B tries to obtain a read or write lock on that record, the thread of control in which Txn B is running is blocked:



However, if Txn A only holds a read lock (a shared lock) on record 002, then only those handles that attempt to obtain a write lock on that record will block.



Blocking and Application Performance

Multi-threaded applications typically perform better than simple single-threaded applications because the application can perform one part of its workload (updating a database record, for example) while it is waiting for some other lengthy operation to complete (performing disk or network I/O, for example). This performance improvement is particularly noticeable if you use hardware that offers multiple CPUs, because the threads can run simultaneously.

That said, concurrent applications can see reduced workload throughput if their threads of control are seeing a large amount of lock contention. That is, if threads are blocking on lock requests, then that represents a performance penalty for your application.

Consider once again the previous diagram of a blocked write lock request. In that diagram, Txn C cannot obtain its requested write lock because Txn A and Txn B are both already holding read locks on the requested record. In this case, the thread in which Txn C is running will pause until such a time as Txn C either obtains its write lock, or the operation that is requesting the lock is abandoned. The fact that Txn C's thread has temporarily halted all forward progress represents a performance penalty for your application.

Moreover, any read locks that are requested while Txn C is waiting for its write lock will also block until such a time as Txn C has obtained and subsequently released its write lock.

Avoiding Blocks

Reducing lock contention is an important part of performance tuning your concurrent JE application. Applications that have multiple threads of control obtaining exclusive (write) locks are prone to contention issues. Moreover, as you increase the numbers of lockers and as you increase the time that a lock is held, you increase the chances of your application seeing lock contention.

As you are designing your application, try to do the following in order to reduce lock contention:

- Reduce the length of time your application holds locks.

Shorter lived transactions will result in shorter lock lifetimes, which will in turn help to reduce lock contention.

In addition, by default transactional cursors hold read locks until such a time as the transaction is completed. For this reason, try to minimize the time you keep transactional cursors opened, or reduce your isolation levels - see below.

- If possible, access heavily accessed (read or write) items toward the end of the transaction. This reduces the amount of time that a heavily used record is locked by the transaction.
- Reduce your application's isolation guarantees.

By reducing your isolation guarantees, you reduce the situations in which a lock can block another lock. Try using uncommitted reads for your read operations in order to prevent a read lock being blocked by a write lock.

In addition, for cursors you can use degree 2 (read committed) isolation, which causes the cursor to release its read locks as soon as it is done reading the record (as opposed to holding its read locks until the transaction ends).

Be aware that reducing your isolation guarantees can have adverse consequences for your application. Before deciding to reduce your isolation, take care to examine your application's isolation requirements. For information on isolation levels, see [Isolation \(page 26\)](#).

- Consider your data access patterns.

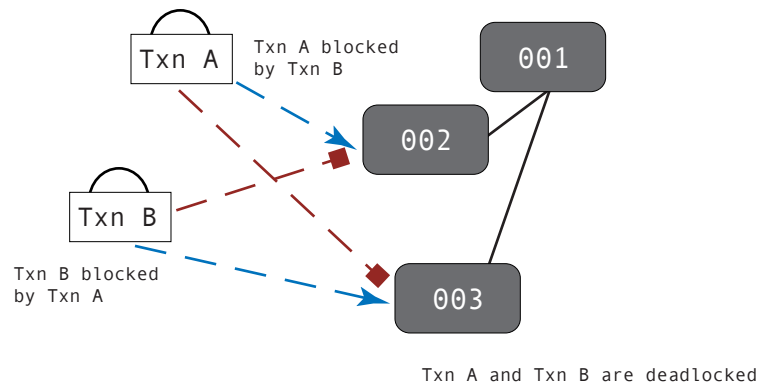
Depending on the nature of your application, this may be something that you can not do anything about. However, if it is possible to create your threads such that they operate only on non-overlapping portions of your database, then you can reduce lock contention because your threads will rarely (if ever) block on one another's locks.

Deadlocks

A deadlock occurs when two or more threads of control are blocked, each waiting on a resource held by the other thread. When this happens, there is no possibility of the threads

ever making forward progress unless some outside agent takes action to break the deadlock.

For example, if Txn A is blocked by Txn B at the same time Txn B is blocked by Txn A then the threads of control containing Txn A and Txn B are *deadlocked*; neither thread can make any forward progress because neither thread will ever release the lock that is blocking the other thread.



When two threads of control deadlock, the only solution is to have a mechanism external to the two threads capable of recognizing the deadlock and notifying at least one thread that it is in a deadlock situation. Once notified, a thread of control must abandon the attempted operation in order to resolve the deadlock. JE is capable of notifying your application when it detects a deadlock. See [Managing Deadlocks \(page 24\)](#) for more information.

Note that when one locker in a thread of control is blocked waiting on a lock held by another locker in that same thread of the control, the thread is said to be *self-deadlocked*.

Note that in JE, a self-deadlock can occur only if two or more transactions (lockers) are used in the same thread. A self-deadlock cannot occur for non-transactional usage, because the thread is the locker.

Deadlock Avoidance

The things that you do to avoid lock contention also help to reduce deadlocks (see [Avoiding Blocks \(page 22\)](#)). Beyond that, you should also make sure all threads access data in the same order as all other threads. So long as threads lock records in the same basic order, there is no possibility of a deadlock (threads can still block, however).

Be aware that if you are using secondary databases (indices), then locking order is different for reading and writing. For this reason, if you are writing a concurrent application and you are using secondary databases, you should expect deadlocks.

JE Lock Management

To manage locks in JE, you must do two things:

1. Manage lock timeouts.
2. Detect and respond to deadlocks.

Managing JE Lock Timeouts

Like transaction timeouts (see [Configuring the Transaction Subsystem \(page 15\)](#)), JE allows you to identify the longest period of time that it is allowed to hold a lock. This value plays an important part in performing deadlock detection, because the only way JE can identify a deadlock is if a lock is held past its timeout value.

However, unlike transaction timeouts, lock timeouts are on a true timer. Transaction timeouts are only identified when JE has a reason to examine its lock table; that is, when it is attempting to acquire a lock. If no such activity is occurring in your application, a transaction can exist for a long time past its expiration timeout. Conversely, lock timeouts are managed by a timer maintained by the JVM. Once this timer has expired, your application will be notified of the event (see the next section on deadlock detection for more information).

You can set the lock timeout on a transaction by transaction basis, or for the entire environment. To set it on a transaction basis, use `Transaction.setLockTimeout()`. To set it for your entire environment, use `EnvironmentConfig.setLockTimeout()` or use the `je.lock.timeout` parameter in the `je.properties` file.

The value that you specify for the lock timeout is in microseconds. 500000 is used by default.

Note that changing this value can have an affect on your application's performance. If you set it too low, locks may expire and be considered deadlocked even though the thread is in fact making forward progress. This will cause your application to abort and retry transactions unnecessarily, which can ultimately harm application throughput. If you set it too high, threads may deadlock for too long before your application receives notification and is able to take corrective action. Again, this can harm application throughput.

Note that for single-threaded applications in which you will have extremely long-lived locks, you may want to set this value to 0. Doing so disables lock timeouts entirely.

Managing Deadlocks

When a lock times out in JE, the thread of control holding that lock is notified of the deadlock event via a `DeadlockException` exception. When this happens, the thread must:

1. Cease all read and write operations.
2. Close all open cursors.

3. Abort the transaction.
4. Optionally retry the operation. If your application retries deadlocked operations, the new attempt must be made using a new transaction.



If a thread has deadlocked, it may not make any additional database calls using the transaction handle that has deadlocked.

For example:

```
// retry_count is a counter used to identify how many times
// we've retried this operation. To avoid the potential for
// endless looping, we won't retry more than MAX_DEADLOCK_RETRIES
// times.

// txn is a transaction handle.
// key and data are DatabaseEntry handles. Their usage is not shown here.
while (retry_count < MAX_DEADLOCK_RETRIES) {
    try {
        txn = myEnv.beginTransaction(null, null);
        myDatabase.put(txn, key, data);
        txn.commit();
        return 0;
    } catch (DeadlockException de) {
        try {
            // Abort the transaction and increment the
            // retry counter
            txn.abort();
            retry_count++;
            if (retry_count >= MAX_DEADLOCK_RETRIES) {
                System.err.println("Exceeded retry limit. Giving up.");
                return -1;
            }
        } catch (DatabaseException ae) {
            System.err.println("txn abort failed: " + ae.toString());
            return -1;
        }
    } catch (DatabaseException e) {
        try {
            // Abort the transaction.
            txn.abort();
        } catch (DatabaseException ae) {
            System.err.println("txn abort failed: " + ae.toString());
        }
        return -1;
    }
}
```


Isolation

Isolation guarantees are an important aspect of transactional protection. Transactions ensure the data your transaction is working with will not be changed by some other transaction. Moreover, the modifications made by a transaction will never be viewable outside of that transaction until the changes have been committed.

That said, there are different degrees of isolation, and you can choose to relax your isolation guarantees to one degree or another depending on your application's requirements. The primary reason why you might want to do this is because of performance; the more isolation you ask your transactions to provide, the more locking that your application must do. With more locking comes a greater chance of blocking, which in turn causes your threads to pause while waiting for a lock. Therefore, by relaxing your isolation guarantees, you can *potentially* improve your application's throughput. Whether you actually see any improvement depends, of course, on the nature of your application's data and transactions.

Supported Degrees of Isolation

JE supports the following levels of isolation:

Degree	ANSI Term	Definition
1	READ UNCOMMITTED	Uncommitted reads means that one transaction will never overwrite another transaction's dirty data. Dirty data is data that a transaction has modified but not yet committed to the underlying data store. However, uncommitted reads allows a transaction to see data dirtied by another transaction. In addition, a transaction may read data dirtied by another transaction, but which subsequently is aborted by that other transaction. In this latter case, the reading transaction may be reading data that never really existed in the database.
2	READ COMMITTED	Committed read isolation means that degree 1 is observed, except that dirty data is never read. In addition, this isolation level guarantees that data will never change so long as it is addressed by the cursor, but the data may change before the reading cursor is closed. In the case of a transaction, data at the current cursor position will not change, but once the cursor moves, the previous referenced data can change. This means that readers release read locks before the cursor is closed, and therefore, before the transaction completes. Note that this level of isolation causes the cursor to operate in exactly the same way as it does in the absence of a transaction.

Degree	ANSI Term	Definition
(undefined)	REPEATABLE READ	Committed read is observed, plus the data read by a transaction, T, will never be dirtied by another transaction before T completes. This means that both read and write locks are not released until the transaction completes. This is JE's default isolation level.
3	SERIALIZABLE	Committed read is observed, plus no transactions will see phantoms. Phantoms are records returned as a result of a search, but which were not seen by the same transaction when the identical search criteria was previously used.

By default, JE transactions and transactional cursors offer repeatable read isolation. You can optionally reduce your isolation level by configuring JE to use uncommitted read isolation. See [Reading Uncommitted Data \(page 27\)](#) for more information. You can also configure JE to use committed read isolation. See [Committed Reads \(page 29\)](#) for more information. Finally, you can configure your transactions and transactional cursors to use serializable isolation. See [Configuring Serializable Isolation \(page 31\)](#) for more information.

Reading Uncommitted Data

You can configure your application to read data that has been modified but not yet committed by another transaction; that is, dirty data. When you do this, you may see a performance benefit by allowing your application to not have to block waiting for write locks. On the other hand, the data that your application is reading may change before the transaction has completed.

When used with transactions, uncommitted reads means that one transaction can see data modified but not yet committed by another transaction. When used with transactional cursors, uncommitted reads means that any database reader can see data modified by the cursor before the cursor's transaction has committed.

Because of this, uncommitted reads allow a transaction to read data that may subsequently be aborted by another transaction. In this case, the reading transaction will have read data that never really existed in the database.

To configure your application to read uncommitted data, specify that you want to use uncommitted reads when you create a transaction or open the cursor. To do this, you use the `setReadUncommitted()` method on the relevant configuration object (`TransactionConfig` or `CursorConfig`).

For example:

```

package je.txn;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import com.sleepycat.je.Transaction;
import com.sleepycat.je.TransactionConfig;

import java.io.File;

...

Database myDatabase = null;
Environment myEnv = null;
try {
    EnvironmentConfig myEnvConfig = new EnvironmentConfig();
    myEnvConfig.setTransactional(true);

    myEnv = new Environment(new File("/my/env/home"),
                           myEnvConfig);

    // Open the database.
    DatabaseConfig dbConfig = new DatabaseConfig();
    dbConfig.setTransactional(true);
    myDatabase = myEnv.openDatabase(null, "sampleDatabase", dbConfig);

    TransactionConfig txnConfig = new TransactionConfig();
    txnConfig.setReadUncommitted(true);           // Use uncommitted reads
                                                    // for this transaction.
    Transaction txn = myEnv.beginTransaction(null, txnConfig);

    // From here, you perform your database reads and writes as normal,
    // committing and aborting the transactions as is necessary, and
    // testing for deadlock exceptions as normal (omitted for brevity).

    ...
}

```

You can also configure uncommitted read isolation on a read-by-read basis by specifying `LockMode.READ_UNCOMMITTED`:

```

package je.txn;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.Environment;
import com.sleepycat.je.LockMode;

```

```

import com.sleepycat.je.Transaction;

...

Database myDb = null;
Environment myEnv = null;
Transaction txn = null;

try {

    // Environment and database open omitted

    ...

    txn = myEnv.beginTransaction(null, null);

    DatabaseEntry theKey =
        new DatabaseEntry((new String("theKey")).getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry();

    myDb.get(txn, theKey, theData, LockMode.READ_UNCOMMITTED);
} catch (Exception e) {
    // Exception handling goes here
}

```

Committed Reads

You can configure your transaction so that the data being read by a transactional cursor is consistent so long as it is being addressed by the cursor. However, once the cursor is done reading the record, the cursor releases its lock on that record. This means that the data the cursor has read and released may change before the cursor's transaction has completed.

For example, suppose you have two transactions, *T_a* and *T_b*. Suppose further that *T_a* has a cursor that reads *record R*, but does not modify it. Normally, *T_b* would then be unable to write *record R* because *T_a* would be holding a read lock on it. But when you configure your transaction for committed reads, *T_b* *can* modify *record R* before *T_a* completes, so long as the reading cursor is no longer addressing the record.

When you configure your application for this level of isolation, you may see better performance throughput because there are fewer read locks being held by your transactions. Read committed isolation is most useful when you have a cursor that is reading and/or writing records in a single direction, and that does not ever have to go back to re-read those same records. In this case, you can allow JE to release read locks as it goes, rather than hold them for the life of the transaction.

To configure your application to use committed reads, do one of the following:

- Create your transaction such that it allows committed reads. You do this by specifying `true` to `TransactionConfig.setReadCommitted()`.
- Specify `true` to `CursorConfig.setReadCommitted()`.

For example, the following creates a transaction that allows committed reads:

```
package je.txn;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import com.sleepycat.je.Transaction;
import com.sleepycat.je.TransactionConfig;

import java.io.File;

...

Database myDatabase = null;
Environment myEnv = null;
try {
    EnvironmentConfig myEnvConfig = new EnvironmentConfig();
    myEnvConfig.setTransactional(true);

    myEnv = new Environment(new File("/my/env/home"),
                           myEnvConfig);

    // Open the database.
    DatabaseConfig dbConfig = new DatabaseConfig();
    dbConfig.setTransactional(true);
    myDatabase = myEnv.openDatabase(null, "sampleDatabase", dbConfig);

    // Open the transaction and enable committed reads. All cursors open
    // with this transaction handle will use read committed isolation.
    TransactionConfig txnConfig = new TransactionConfig();
    txnConfig.setReadCommitted(true);           // Use committed reads
                                                // for this transaction.
    Transaction txn = myEnv.beginTransaction(null, txnConfig);

    // From here, you perform your database reads and writes as normal,
    // committing and aborting the transactions as is necessary, and
    // testing for deadlock exceptions as normal (omitted for brevity).

    ...
}
```

You can also configure read committed isolation on a read-by-read basis by specifying `LockMode.READ_COMMITTED`:

```
package je.txn;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.Environment;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.Transaction;

...

Database myDb = null;
Environment myEnv = null;
Transaction txn = null;

try {

    // Environment and database open omitted

    ...

    txn = myEnv.beginTransaction(null, null);

    DatabaseEntry theKey =
        new DatabaseEntry((new String("theKey")).getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry();

    myDb.get(txn, theKey, theData, LockMode.READ_COMMITTED);
} catch (Exception e) {
    // Exception handling goes here
}
```

Configuring Serializable Isolation

You can configure JE to use serializable isolation. Serializable isolation prevents transactions from seeing *phantoms*. Phantoms occur when a transaction obtains inconsistent results when performing a given query.

Suppose a transaction performs a search, *S*, and as a result of that search NOTFOUND is returned. If you are using only repeatable read isolation (the default isolation level), it is possible for the same transaction to perform *S* at a later point in time and return SUCCESS instead of NOTFOUND. This can occur if another thread of control modified the database in such a way as to cause *S* to successfully locate data, where before no data was found. When this situation occurs, the results returned by *S* are said to be a *phantom*.

To prevent phantoms, you can use serializable isolation. Note that this causes JE to perform additional locking in order to prevent keys from being inserted until the

transaction ends. However, this additional locking can also result in reduced concurrency for your application, which means that your database access can be slowed.

You configure serializable isolation for all transactions in your environment by using

`EnvironmentConfig.setTxnSerializableIsolation()`:

```
package je.txn;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import com.sleepycat.je.Transaction;
import com.sleepycat.je.LockMode;

...

Database myDb = null;
Environment myEnv = null;
Transaction txn = null;

try {

    // Open an environment
    EnvironmentConfig envConfig = new EnvironmentConfig();
    envConfig.setAllowCreate(true);
    envConfig.setTransactional(true);

    // Use serializable isolation
    envConfig.setTxnSerializableIsolation(true);

    myEnv = new Environment(myHomeDirectory, envConfig);

    // Database open omitted

    ...

    txn = myEnv.beginTransaction(null, null);

    DatabaseEntry theKey =
        new DatabaseEntry((new String("theKey")).getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry();

    myDb.get(txn, theKey, theData, LockMode.DEFAULT);
} catch (Exception e) {
    // Exception handling goes here
}
```

If you do not configure serializable isolation for all transactions, you can configure serializable isolation for a specific transaction using

`TransactionConfig.setSerializableIsolation()`:

```
package je.txn;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.Environment;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.Transaction;
import com.sleepycat.je.TransactionConfig;

...

Database myDb = null;
Environment myEnv = null;
Transaction txn = null;

try {

    // Environment and database open omitted

    ...

    TransactionConfig tc = new TransactionConfig();
    tc.setSerializableIsolation(true); // Use serializable isolation
    txn = myEnv.beginTransaction(null, tc);

    DatabaseEntry theKey =
        new DatabaseEntry((new String("theKey")).getBytes("UTF-8"));
    DatabaseEntry theData = new DatabaseEntry();

    myDb.get(txn, theKey, theData, LockMode.DEFAULT);
} catch (Exception e) {
    // Exception handling goes here
}
```

Transactional Cursors and Concurrent Applications

When you use transactional cursors with a concurrent application, remember that in the event of a deadlock you must make sure that you close your cursor before you abort and retry your transaction.

Also, remember that when you are using the default isolation level, every time your cursor reads a record it locks that record until the encompassing transaction is resolved. This means that walking your database with a transactional cursor increases the chance of lock contention.

For this reason, if you must routinely walk your database with a transactional cursor, consider using a reduced isolation level such as read committed.

Using Cursors with Uncommitted Data

As described in [Reading Uncommitted Data \(page 27\)](#) above, it is possible to relax your transaction's isolation level such that it can read data modified but not yet committed by another transaction. You can configure this when you create your transaction handle, and when you do so then all cursors opened inside that transaction will automatically use uncommitted reads.

You can also do this when you create a cursor handle from within a serializable transaction. When you do this, only those cursors configured for uncommitted reads uses uncommitted reads.

The following example shows how to configure an individual cursor handle to read uncommitted data from within a serializable (full isolation) transaction. For an example of configuring a transaction to perform uncommitted reads in general, see [Reading Uncommitted Data \(page 27\)](#).

```
package je.txn;

import com.sleepycat.je.Cursor;
import com.sleepycat.je.CursorConfig;
import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;

import java.io.File;

...

Database myDatabase = null;
Environment myEnv = null;
try {

    EnvironmentConfig myEnvConfig = new EnvironmentConfig();
    myEnvConfig.setTransactional(true);

    myEnv = new Environment(new File("/my/env/home"),
                           myEnvConfig);

    // Open the database.
    DatabaseConfig dbConfig = new DatabaseConfig();
    dbConfig.setTransactional(true);
    myDatabase = myEnv.openDatabase(null,           // txn handle
                                   "sampleDatabase", // db file name
                                   dbConfig);
```

```
// Open the transaction. Note that this is a repeatable
// read transaction.
Transaction txn = myEnv.beginTransaction(null, null);
Cursor cursor = null;
try {
    // Use the transaction handle here
    // Get our cursor. Note that we pass the transaction
    // handle here. Note also that we cause the cursor
    // to perform uncommitted reads.
    CursorConfig cconfig = new CursorConfig();
    cconfig.setReadUncommitted(true);
    cursor = db.openCursor(txn, cconfig);

    // From here, you perform your cursor reads and writes
    // as normal, committing and aborting the transactions as
    // is necessary, and testing for deadlock exceptions as
    // normal (omitted for brevity).

    ...
}
```

Read/Modify/Write

If you are retrieving a record from the database for the purpose of modifying or deleting it, you should declare a read-modify-write cycle at the time that you read the record. Doing so causes JE to obtain write locks (instead of a read locks) at the time of the read. This helps to prevent deadlocks by preventing another transaction from acquiring a read lock on the same record while the read-modify-write cycle is in progress.

Note that declaring a read-modify-write cycle may actually increase the amount of blocking that your application sees, because readers immediately obtain write locks and write locks cannot be shared. For this reason, you should use read-modify-write cycles only if you are seeing a large amount of deadlocking occurring in your application.

In order to declare a read/modify/write cycle when you perform a read operation, specify `com.sleepycat.je.LockMode.RMW` to the database or cursor get method.

For example:

```
// Begin the deadlock retry loop as is normal.
while (retry_count < MAX_DEADLOCK_RETRIES) {
    try {
        txn = myEnv.beginTransaction(null, null);

        ...
        // key and data are DatabaseEntry objects.
        // Their usage is omitted for brevity.
        ...

        // Read the data. Declare the read/modify/write cycle here
        myDatabase.get(txn, key, data, LockMode.RMW);
    }
}
```

```
    // Put the data. Note that you do not have to provide any
    // additional flags here due to the read/modify/write
    // cycle. Simply put the data and perform your deadlock
    // detection as normal.
    myDatabase.put(txn, key, data);
    txn.commit();
    return 0;
} catch (DeadlockException de) {
    // Deadlock detection and exception handling omitted
    // for brevity
    ...
}
```

Chapter 5. Backing up and Restoring Berkeley DB, Java Edition Applications

Fundamentally, you backup your databases by copying JE log files off to a safe storage location. To restore your database from a backup, you copy those files to an appropriate directory on disk and restart your JE application.

Beyond these simple activities, there are some differing backup strategies that you may want to consider. These topics are described in this chapter.

Before continuing, before you review the information on log files and background threads in the *Getting Started with Berkeley DB, Java Edition*. Those topics contain important information that is basic to the following discussion on backups and restores.

Normal Recovery

Remember that internally JE databases are organized in a BTree, and that in order to operate JE requires the complete BTree be available to it.

When database records are created, modified, or deleted, the modifications are represented in the BTree's leaf nodes. Beyond leaf node changes, database record modifications can also cause changes to other BTree nodes and structures.

Now, if your writes are transaction-protected, then every time a transaction is committed the leaf nodes (and *only* the leaf nodes) modified by that transaction are written to the JE log files on disk. Also, remember that the durability of the write (whether a flush or fsync is performed) depends on the type of commit that is requested. See [Non-Durable Transactions \(page 9\)](#) for more information.

Normal recovery, then, is the process of recreating the entire BTree from the information available in the leaf nodes. You do not have to do anything special to cause normal recovery to be run; this occurs every time a JE environment is opened.

Checkpoints

Running normal recovery can become expensive if over time all that is ever written to disk is BTree leaf nodes. So in order to limit the time required for normal recovery, JE runs checkpoints. Checkpoints write to your log files all the internal BTree nodes and structures modified as a part of write operations. This means that your log files contain a complete BTree up to the moment in time when the checkpoint was run. The result is that normal recovery only needs to recreate the portion of the BTree that has been modified since the time of the last checkpoint.

Checkpoints typically write more information to disk than do transaction commits, and so they are more expensive from a disk I/O perspective. You will therefore need to consider how frequently to run checkpoints as a part of your performance tuning activities. When you do this, balance the cost of the checkpoints against the time it will take your application to restart due to the cost of running normal recovery.

Checkpoints are normally performed by the checkpointer background thread, which is always running. Like all background threads, it is managed using the `je.properties` file. Currently, the only checkpointer property that you may want to manage is `je.checkpointer.bytesInterval`. This property identifies how much JE's log files can grow before a checkpoint is run. Its value is specified in bytes. Decreasing this value causes the checkpointer thread to run checkpoints more frequently. This will improve the time that it takes to run recovery, but it also increases the system resources (notably, I/O) required by JE.

Note that checkpoints are also always performed when the environment is closed normally. Therefore, normal recovery only has work to do if the application crashes or otherwise ends abnormally without calling `Environment.close()`.

Performing Backups

This section describes how to backup your JE database(s) such that catastrophic recovery is possible.

To backup your database, you can either take a hot backup or an offline backup. A hot backup is performed while database write operations are in progress.

Do not confuse offline and hot backups with the concept of a full and incremental backup. Both an offline and a hot backup are full backups - you back up the entire database. The only difference between them is how much of the contents of the in-memory cache are contained in them. On the other hand, an incremental backup is a backup of just those log files modified or created since the time of the last backup. Most backup software is capable of performing both full and incremental backups for you.

Performing a Hot Backup

To perform a hot backup of your JE databases, copy all log files (*.jdb files) from your environment directory to your archival location or backup media. The files must be copied in alphabetical order (numerical in effect). You do not have to stop any database operations in order to do this.

To make this process a bit easier, you may want to make use of the `DbBackup` helper class. See [Using the DbBackup Helper Class \(page 39\)](#) for details.

Performing an Offline Backup

An offline backup guarantees that you have captured the database in its entirety, including all contents of your in-memory cache, at the moment that the backup was taken. To do this, you must make sure that no write operations are in progress and all database modifications have been written to your log files on disk. To obtain an offline backup:

1. Stop writing your databases.
2. Make sure all your in-memory changes have been flushed to disk. How you do this depends on the type of transactions that you are using:

- If you are using transactions that writes all dirty data to disk on commit (this is the default behavior), you simply need to make sure all on-going transactions are committed or aborted.
- If you are using transactions that do not synchronously write on commit, you must run a checkpoint. Remember that closing your environment causes a checkpoint to be run, so if your application is shutting down completely before taking the backup, you have met this requirement.

For information on changing the transactional sync behavior, see [Non-Durable Transactions \(page 9\)](#). For information on running a checkpoint, see [Checkpoints \(page 37\)](#).

3. If you are using durable transactions, then optionally run a checkpoint. Doing this can shorten the time required to restore your database from this back up.
4. Copy all log files (*.jdb) from your environment directory to your archival location or backup media. To make this process a bit easier, you may want to make use of the DbBackup helper class. See the next section for details.

You can now resume normal database operations.

Using the DbBackup Helper Class

In order to simplify backup operations, JE provides the DbBackup helper class. This class stops and restarts JE background activity in an open environment. It also lets the application create a backup which can support restoring the environment to a specific point in time.

Because you do not have to stop JE write activity in order to take a backup, it is usually necessary to examine your log files twice before you decide that your backup is complete. This is because JE may create a new log file while you are running your backup. A second pass over your log files allows you to ensure that no new files have been created and so you can declare your backup complete.

For example:

time	files in environment	activity
t0	000000001.jdb 000000003.jdb 000000004.jdb	Backup starts copying file 1
t1	000000001.jdb 000000004.jdb 000000005.jdb	JE log cleaner migrates portion of file 3 to newly created file 5 and deletes file 3. Backup finishes file 1, starts copying file 4. Backup MUST include file 5 for a consistent backup!
t2	000000001.jdb	Backup finishes copying file 4, starts and finishes

```
000000004.jdb    file 5, has caught up. Backup ends.  
000000005.jdb
```

DbBackup works around this problem by defining the set of files that must be copied for each backup operation, and freezes all changes to those files. The application can copy that defined set of files and finish operation without checking for the ongoing creation of new files. Also, there will be no need to check for a newer version of the last file on the next backup.

In the example above, if DbBackup was used at t0, the application would only have to copy files 1, 3 and 4 to back up. On a subsequent backup, the application could start its copying at file 5. There would be no need to check for a newer version of file 4.

The following code fragment illustrates this class' usage:

```
package je.gettingStarted;  
  
...  
import com.sleepycat.je.util.DbBackup;  
...  
  
Environment env = new Environment(...);  
DbBackup backupHelper = new DbBackup(env);  
  
// Find the file number of the last file in the previous backup  
// persistently, by either checking the backup archive, or saving  
// state in a persistent file.  
long lastFileCopiedInPrevBackup = ...  
  
// Start backup, find out what needs to be copied.  
backupHelper.startBackup();  
try {  
    String filesForBackup =  
        backupHelper.getLogFilesInBackupSet(lastFileCopiedInPrevBackup);  
  
    // Copy the files to archival storage.  
    myApplicationCopyMethod(filesForBackup)  
    // Update our knowledge of the last file saved in the backup set,  
    // so we can copy less on the next backup  
    lastFileCopiedInPrevBackup = backupHelper.getLastFileInBackupSet();  
    myApplicationSaveLastFile(lastFileCopiedInBackupSet);  
}  
finally {  
    // Remember to exit backup mode, or all log files won't be cleaned  
    // and disk usage will bloat.  
    backupHelper.endBackup();  
}
```

Performing Catastrophic Recovery

Catastrophic recovery is necessary whenever your environment and/or database have been lost or corrupted due to a media failure (disk failure, for example). Catastrophic recovery is also required if normal recovery fails for any reason.

In order to perform catastrophic recovery, you must have a full backup of your databases. You will use this backup to restore your database. See the previous section for information on running back ups.

To perform catastrophic recovery:

1. Shut down your application.
2. Delete the contents of your environment home directory (the one that experienced a catastrophic failure), if there is anything there.
3. Copy your most recent full backup into your environment home directory.
4. If you are using a backup utility that runs incremental backups of your environment directory, copy any log files generated since the time of your last full backup. Be sure to restore all log files in the order that they were written. The order is important because it is possible the same log file appears in multiple archives, and you want to run recovery using the most recent version of each log file.
5. Open the environment as normal. JE's normal recovery will run, which will bring your database to a consistent state relative to the changed data found in your log files.

You are now done restoring your database.

Hot Failover

As a final backup/recovery strategy, you can create a hot failover. Note that using hot failovers requires your application to be able to specify its environment home directory at application startup time. Most application developers allow the environment home directory to be identified using a command line option or a configuration or properties file. If your application has its environment home hard-coded into it, you cannot use hot failovers.

You create a hot failover by periodically backing up your database to an alternative location on disk. Usually this alternative location is on a separate physical drive from where you normally keep your database, but if multiple drives are not available then you should at least put the hot failover on a separate disk partition.

You failover by causing your application to reopen its environment using the failover location.

Note that a hot failover should not be used as a substitute for backing up and archiving your data to a safe location physically remote from your computing environment. Even if your data is spread across multiple physical disks, a truly serious catastrophe (fires,

malevolent software viruses, faulty disk controllers, and so forth) can still cause you to lose your data.

To create and maintain a hot failover:

1. Copy all log files (*.jdb) from your environment directory to the location where you want to keep your failover. Either an offline or a hot backup can be used for this purpose, but typically a hot failover is initially created by taking an offline backup of your database. This ensures that you have captured the contents of your in-memory cache.
2. Periodically copy to your failover directory any log files that were changed or created since the time of your last copy. Most backup software is capable of performing this kind of an incremental backup for you.

Note that the frequency of your incremental copies determines the amount of data that is at risk due to catastrophic failures. For example, if you perform the incremental copy once an hour then at most your hot failover is an hour behind your production database, and so you are risking at most an hour's worth of database changes.

3. Remove any *.jdb files from the hot failover directory that have been removed or renamed to .del files in the primary directory. This is not necessary for consistency, but will help to reduce disk space consumed by the hot failover.

Chapter 6. Summary and Examples

Throughout this manual we have presented the concepts and mechanisms that you need to provide transactional protection for your application. In this chapter, we summarize these mechanisms, and we provide a complete example of a multi-threaded transactional JE application.

Anatomy of a Transactional Application

Transactional applications are characterized by performing the following activities:

1. Create your environment handle.
2. Open your environment, specifying that the transactional subsystem is to be used.
3. Open your database handles, indicating that they are to support transactions.
4. Spawn off worker threads. How many of these you need and how they split their JE workload is entirely up to your application's requirements. However, any worker threads that perform write operations against your databases will do the following:
 - a. Begin a transaction.
 - b. Perform one or more read and write operations against your databases.
 - c. Commit the transaction if all goes well.
 - d. Abort and retry the operation if a deadlock is detected.
 - e. Abort the transaction for most other errors.
5. On application shutdown:
 - a. Make sure there are no opened cursors.
 - b. Make sure there are no active transactions. Either abort or commit all transactions before shutting down.
 - c. Close your databases
 - d. Close your environment.



Robust applications should monitor their database worker threads to make sure they have not died unexpectedly. If a thread does terminate abnormally, you must shutdown all your worker threads and then run normal recovery (you will have to reopen your environment to do this). This is the only way to clear any resources (such as a lock or a mutex) that the abnormally exiting worker thread might have been holding at the time that it died.

Failure to perform this recovery can cause your still-functioning worker threads to eventually block forever while waiting for a lock that will never be released.

In addition to these activities, which are entirely handled by code within your application, you also need to periodically back up your log files. This is required in order to obtain the durability guarantee made by JE's transaction ACID support. See [Backing up and Restoring Berkeley DB, Java Edition Applications \(page 37\)](#) for more information.

Transaction Example

The following Java code provides a fully functional example of a multi-threaded transactional JE application. The example opens an environment and database, and then creates 5 threads, each of which writes 500 records to the database. The keys used for these writes are pre-determined strings, while the data is a class that contains randomly generated data. This means that the actual data is arbitrary and therefore uninteresting; we picked it only because it requires minimum code to implement and therefore will stay out of the way of the main points of this example.

Each thread writes 10 records under a single transaction before committing and writing another 10 (this is repeated 50 times). At the end of each transaction, but before committing, each thread calls a function that uses a cursor to read every record in the database. We do this in order to make some points about database reads in a transactional environment.

Of course, each writer thread performs deadlock detection as described in this manual. In addition, normal recovery is performed when the environment is opened.

To implement this example, we need three classes:

- `TxnGuide.java`

This is the main class for the application. It performs environment and database management, spawns threads, and creates the data that is placed in the database. See [TxnGuide.java \(page 45\)](#) for implementation details.

- `DBWriter.java`

This class extends `java.lang.Thread`, and as such it is our thread implementation. It is responsible for actually reading and writing to the database. It also performs all of our transaction management. See [DBWriter.java \(page 49\)](#) for implementation details.

- `PayloadData.java`

This is a data class used to encapsulate several data fields. It is fairly uninteresting, except that the usage of a class means that we have to use the bind APIs to serialize it for storage in the database. See [PayloadData.java \(page 48\)](#) for implementation details.

TxnGuide.java

The main class in our example application is used to open and close our environment and database. It also spawns all the threads that we need. We start with the normal series of Java package and import statements, followed by our class declaration:

```
// File TxnGuide.java

package je.txn;

import com.sleepycat.bind.serial.StoredClassCatalog;

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseConfig;
import com.sleepycat.je.DatabaseException;

import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;

import java.io.File;
import java.io.FileNotFoundException;

public class TxnGuide {
```

Next we declare our class' private data members. Mostly these are used for constants such as the name of the database that we are opening and the number of threads that we are spawning. However, we also declare our environment and database handles here.

```
private static String myEnvPath = "./";
private static String dbName = "mydb.db";
private static String cdbName = "myclassdb.db";

// DB handles
private static Database myDb = null;
private static Database myClassDb = null;
private static Environment myEnv = null;

private static final int NUMTHREADS = 5;
```

Next, we implement our `usage()` method. This application optionally accepts a single command line argument which is used to identify the environment home directory.

```
private static void usage() {
    System.out.println("TxnGuide [-h <env directory>]");
    System.exit(-1);
}
```

Now we implement our `main()` method. This method simply calls the methods to parse the command line arguments and open the environment and database. It also creates the stored class catalog that we use for serializing the data that we want to store in our database. Finally, it creates and then joins the database writer threads.

```

public static void main(String args[]) {
    try {
        // Parse the arguments list
        parseArgs(args);
        // Open the environment and databases
        openEnv();
        // Get our class catalog (used to serialize objects)
        StoredClassCatalog classCatalog =
            new StoredClassCatalog(myClassDb);

        // Start the threads
        DBWriter[] threadArray;
        threadArray = new DBWriter[NUMTHREADS];
        for (int i = 0; i < NUMTHREADS; i++) {
            threadArray[i] = new DBWriter(myEnv, myDb, classCatalog);
            threadArray[i].start();
        }

        // Join the threads. That is, wait for each thread to
        // complete before exiting the application.
        for (int i = 0; i < NUMTHREADS; i++) {
            threadArray[i].join();
        }
    } catch (Exception e) {
        System.err.println("TxnGuide: " + e.toString());
        e.printStackTrace();
    } finally {
        closeEnv();
    }
    System.out.println("All done.");
}

```

Next we implement `openEnv()`. This method is used to open the environment and then a database in that environment. Along the way, we make sure that the transactional subsystem is correctly initialized.

For the database open, notice that we open the database such that it supports duplicate records. This is required purely by the data that we are writing to the database, and it is only necessary if you run the application more than once without first deleting the environment.

```

private static void openEnv() throws DatabaseException {
    System.out.println("opening env");

    // Set up the environment.
    EnvironmentConfig myEnvConfig = new EnvironmentConfig();
    myEnvConfig.setAllowCreate(true);
    myEnvConfig.setTransactional(true);
    // Environment handles are free-threaded by default in JE,
    // so we do not have to do anything to cause the

```

```

// environment handle to be free-threaded.

// Set up the database
DatabaseConfig myDbConfig = new DatabaseConfig();
myDbConfig.setAllowCreate(true);
myDbConfig.setTransactional(true);
myDbConfig.setSortedDuplicates(true);

// Open the environment
myEnv = new Environment(new File(myEnvPath),    // Env home
                        myEnvConfig);

// Open the database. Do not provide a txn handle. This open
// is auto committed because DatabaseConfig.setTransactional()
// is true.
myDb = myEnv.openDatabase(null,    // txn handle
                          dbName,  // Database file name
                          myDbConfig);

// Used by the bind API for serializing objects
// Class database must not support duplicates
myDbConfig.setSortedDuplicates(false);
myClassDb = myEnv.openDatabase(null,    // txn handle
                              cdbName, // Database file name
                              myDbConfig);
}

```

Finally, we implement the methods used to close our environment and databases, parse the command line arguments, and provide our class constructor. This is fairly standard code and it is mostly uninteresting from the perspective of this manual. We include it here purely for the purpose of completeness.

```

private static void closeEnv() {
    System.out.println("Closing env and databases");
    if (myDb != null ) {
        try {
            myDb.close();
        } catch (DatabaseException e) {
            System.err.println("closeEnv: myDb: " +
                               e.toString());
            e.printStackTrace();
        }
    }

    if (myClassDb != null ) {
        try {
            myClassDb.close();
        } catch (DatabaseException e) {
            System.err.println("closeEnv: myClassDb: " +

```

```

        e.toString();
        e.printStackTrace();
    }
}

if (myEnv != null ) {
    try {
        myEnv.close();
    } catch (DatabaseException e) {
        System.err.println("closeEnv: " + e.toString());
        e.printStackTrace();
    }
}

private TxnGuide() {}

private static void parseArgs(String args[]) {
    for(int i = 0; i < args.length; ++i) {
        if (args[i].startsWith("-")) {
            switch(args[i].charAt(1)) {
                case 'h':
                    myEnvPath = new String(args[++i]);
                    break;
                default:
                    usage();
            }
        }
    }
}
}
}
}

```

PayloadData.java

Before we show the implementation of the database writer thread, we need to show the class that we will be placing into the database. This class is fairly minimal. It simply allows you to store and retrieve an int, a String, and a double. We will be using the Sleepycat bind API from within the writer thread to serialize instances of this class and place them into our database.

```

package je.txn;

import java.io.Serializable;

public class PayloadData implements Serializable {
    private int oID;
    private String threadName;
    private double doubleData;
}

```

```

    PayloadData(int id, String name, double data) {
        oID = id;
        threadName = name;
        doubleData = data;
    }

    public double getDoubleData() { return doubleData; }
    public int getID() { return oID; }
    public String getThreadName() { return threadName; }
}

```

DBWriter.java

DBWriter.java provides the implementation for our database writer thread. It is responsible for:

- All transaction management.
- Responding to deadlock exceptions.
- Providing data to be stored into the database.
- Serializing and then writing the data to the database.

In order to show off some of the ACID properties provided by JE's transactional support, DBWriter.java does some things in a less efficient way than you would probably decide to use in a true production application. First, it groups 10 database writes together in a single transaction when you could just as easily perform one write for each transaction. If you did this, you could use auto commit for the individual database writes, which means your code would be slightly simpler and you would run a *much* smaller chance of encountering blocked and deadlocked operations. However, by doing things this way, we are able to show transactional atomicity, as well as deadlock handling.

At the end of each transaction, DBWriter.java runs a cursor over the entire database by way of counting the number of records currently existing in the database. There are better ways to discover this information, but in this case we want to make some points regarding cursors, transactional applications, and deadlocking (we get into this in more detail later in this section).

To begin, we provide the usual package and import statements, and we declare our class:

```

package je.txn;

import com.sleepycat.bind.EntryBinding;
import com.sleepycat.bind.serial.StoredClassCatalog;
import com.sleepycat.bind.serial.SerialBinding;
import com.sleepycat.bind.tuple.StringBinding;

import com.sleepycat.je.Cursor;
import com.sleepycat.je.CursorConfig;

```



```

import com.sleepycat.je.Database;
import com.sleepycat.je.DatabaseEntry;
import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.DeadlockException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.LockMode;
import com.sleepycat.je.OperationStatus;
import com.sleepycat.je.Transaction;

import java.io.UnsupportedEncodingException;
import java.util.Random;

public class DBWriter extends Thread
{

```

Next we declare our private data members. Notice that we get handles for the environment and the database. We also obtain a handle for an `EntryBinding`. We will use this to serialize `PayloadData` class instances (see [PayloadData.java \(page 48\)](#)) for storage in the database. The random number generator that we instantiate is used to generate unique data for storage in the database. The `MAX_RETRY` variable is used to define how many times we will retry a transaction in the face of a deadlock. And, finally, `keys` is a `String` array that holds the keys used for our database entries.

```

    private Database myDb = null;
    private Environment myEnv = null;
    private EntryBinding dataBinding = null;
    private Random generator = new Random();

    private static final int MAX_RETRY = 20;

    private static String[] keys = {"key 1", "key 2", "key 3",
                                    "key 4", "key 5", "key 6",
                                    "key 7", "key 8", "key 9",
                                    "key 10"};

```

Next we implement our class constructor. The most interesting thing we do here is instantiate a serial binding for serializing `PayloadData` instances.

```

    // Constructor. Get our DB handles from here
    DBWriter(Environment env, Database db, StoredClassCatalog scc)
        throws DatabaseException {
        myDb = db;
        myEnv = env;
        dataBinding = new SerialBinding(scc, PayloadData.class);
    }

```

Now we implement our thread's `run()` method. This is the method that is run when `DBWriter` threads are started in the main program (see [TxnGuide.java \(page 45\)](#)).

```
// Thread method that writes a series of records
// to the database using transaction protection.
// Deadlock handling is demonstrated here.
public void run () {
```

The first thing we do is get a null transaction handle before going into our main loop. We also begin the top transaction loop here that causes our application to perform 50 transactions.

```
Transaction txn = null;

// Perform 50 transactions
for (int i=0; i<50; i++) {
```

Next we declare a `retry` variable. This is used to determine whether a deadlock should result in our retrying the operation. We also declare a `retry_count` variable that is used to make sure we do not retry a transaction forever in the unlikely event that the thread is unable to ever get a necessary lock. (The only thing that might cause this is if some other thread dies while holding an important lock. This is the only code that we have to guard against that because the simplicity of this application makes it highly unlikely that it will ever occur.)

```
boolean retry = true;
int retry_count = 0;
// while loop is used for deadlock retries
while (retry) {
```

Now we go into the `try` block that we use for deadlock detection. We also begin our transaction here.

```
// try block used for deadlock detection and
// general db exception handling
try {

    // Get a transaction
    txn = myEnv.beginTransaction(null, null);
```

Now we write 10 records under the transaction that we have just begun. By combining multiple writes together under a single transaction, we increase the likelihood that a deadlock will occur. Normally, you want to reduce the potential for a deadlock and in this case the way to do that is to perform a single write per transaction. In other words, we *should* be using auto commit to write to our database for this workload.

However, we want to show deadlock handling and by performing multiple writes per transaction we can actually observe deadlocks occurring. We also want to underscore the idea that you can combine multiple database operations together in a single atomic unit of work. So for our example, we do the (slightly) wrong thing.

Further, notice that we store our key into a `DatabaseEntry` using `com.sleepycat.bind.tuple.StringBinding` to perform the serialization. Also, when we instantiate the `PayloadData` object, we call `getName()` which gives us the string

representation of this thread's name, as well as `Random.nextDouble()` which gives us a random double value. This latter value is used so as to avoid duplicate records in the database.

```
// Write 10 records to the db
// for each transaction
for (int j = 0; j < 10; j++) {
    // Get the key
    DatabaseEntry key = new DatabaseEntry();
    StringBinding.stringToEntry(keys[j], key);

    // Get the data
    PayloadData pd = new PayloadData(i+j, getName(),
        generator.nextDouble());
    DatabaseEntry data = new DatabaseEntry();
    dataBinding.objectToEntry(pd, data);

    // Do the put
    myDb.put(txn, key, data);
}
```

Having completed the inner database write loop, we could simply commit the transaction and continue on to the next block of 10 writes. However, we want to first illustrate a few points about transactional processing so instead we call our `countRecords()` method before calling the transaction commit. `countRecords()` uses a cursor to read every record in the database and return a count of the number of records that it found.

Because `countRecords()` reads every record in the database, if used incorrectly the thread will self-deadlock. The writer thread has just written 500 records to the database, but because the transaction used for that write has not yet been committed, each of those 500 records are still locked by the thread's transaction. If we then simply run a non-transactional cursor over the database from within the same thread that has locked those 500 records, the cursor will block when it tries to read one of those transactional protected records. The thread immediately stops operation at that point while the cursor waits for the read lock it has requested. Because that read lock will never be released (the thread can never make any forward progress), this represents a self-deadlock for the thread.

There are three ways to prevent this self-deadlock:

1. We can move the call to `countRecords()` to a point after the thread's transaction has committed.
2. We can allow `countRecords()` to operate under the same transaction as all of the writes were performed.
3. We can reduce our isolation guarantee for the application by allowing uncommitted reads.

For this example, we choose to use option 3 (uncommitted reads) to avoid the deadlock. This means that we have to open our cursor handle so that it knows to perform uncommitted reads.

```
// commit
System.out.println(getName() + " : committing txn : "
    + i);

// Using uncommitted reads to avoid the deadlock, so null
// is passed for the transaction here.
System.out.println(getName() + " : Found " +
    countRecords(null) + " records in the database.");
```

Having performed this somewhat inelegant counting of the records in the database, we can now commit the transaction.

```
try {
    txn.commit();
    txn = null;
} catch (DatabaseException e) {
    System.err.println("Error on txn commit: " +
        e.toString());
}
retry = false;
```

If all goes well with the commit, we are done and we can move on to the next batch of 10 records to add to the database. However, in the event of an error, we must handle our exceptions correctly. The first of these is a deadlock exception. In the event of a deadlock, we want to abort and retry the transaction, provided that we have not already exceeded our retry limit for this transaction.

```
} catch (DeadlockException de) {
    System.out.println("##### " + getName() +
        " : caught deadlock");
    // retry if necessary
    if (retry_count < MAX_RETRY) {
        System.err.println(getName() +
            " : Retrying operation.");
        retry = true;
        retry_count++;
    } else {
        System.err.println(getName() +
            " : out of retries. Giving up.");
        retry = false;
    }
}
```

In the event of a standard, non-specific database exception, we simply log the exception and then give up (the transaction is not retried).

```
} catch (DatabaseException e) {
    // abort and don't retry
```

```

        retry = false;
        System.err.println(getName() +
            " : caught exception: " + e.toString());
        e.printStackTrace();

```

And, finally, we always abort the transaction if the transaction handle is not null. Note that immediately after committing our transaction, we set the transaction handle to null to guard against aborting a transaction that has already been committed.

```

        } finally {
            if (txn != null) {
                try {
                    txn.abort();
                } catch (Exception e) {
                    System.err.println("Error aborting txn: " +
                        e.toString());
                    e.printStackTrace();
                }
            }
        }
    }
}

```

The final piece of our `DBWriter` class is the `countRecords()` implementation. Notice how in this example we open the cursor such that it performs uncommitted reads:

```

// A method that counts every record in the database.

// Note that this method exists only for illustrative purposes.
// A more straight-forward way to count the number of records in
// a database is to use the Database.getStats() method.
private int countRecords(Transaction txn) throws DatabaseException {
    DatabaseEntry key = new DatabaseEntry();
    DatabaseEntry data = new DatabaseEntry();
    int count = 0;
    Cursor cursor = null;

    try {
        // Get the cursor
        CursorConfig cc = new CursorConfig();
        cc.setReadUncommitted(true);
        cursor = myDb.openCursor(txn, cc);
        while (cursor.getNext(key, data, LockMode.DEFAULT) ==
            OperationStatus.SUCCESS) {

            count++;
        }
    } finally {
        if (cursor != null) {

```

```
        cursor.close();
    }
}

return count;
}
}
```

This completes our transactional example. If you would like to experiment with this code, you can find the example in the following location in your JE distribution:

```
JE_HOME/examples/je/txn
```